SecureDB - A Secure Query Processing System with data interoperability

Supervisor: Professor Benjamin Kao

Group Member: Lin Haibin
Table of Contents

1. Introduction 3
2. Project Background and Literature Review 4
3. Project Methodology 6
   3.1 Theoretical Background 6
      3.1.1 Encryption Procedure 6
      3.1.2 Secure Operators 7
   3.2 System Architecture 10
      3.2.1 Overall Architecture 10
      3.2.2 System Components 11
   3.3 Query Rewriting 14
      3.3.1 Query Rewriting for Multiplication 14
      3.3.2 Query Rewriting for Addition 15
      3.3.3 Query Rewriting for Subtraction 15
      3.3.4 Query Rewriting for Comparison 16
   3.4 Web Interface 18
      3.4.1 Web Interface for Data Upload 18
      3.4.1 Web Interface for Query 19
4. Experiments and Results 21
   4.1 Experiment Environment 21
   4.2 Experiment Result 22
5. Conclusion and Future Works 25
   5.1 Secure Operator Extension 25
   5.2 Cryptographic Optimization 25
   5.3 Conclusion 26
6. References 27
7. Appendix 28
   7.1 Rewrite Rules for Other Secure Operators 28
   7.2 Experiment Data 28
1. Introduction

As Database-as-a-service (DBaaS) such as Amazon Web Service and Microsoft Azure gets more economically available, security issue in the public cloud becomes one of the major concerns of data owners (DO). Despite the advantages of computation elasticity and scalability provided by the cloud service provider (SP), data owner cannot afford the risk of having highly sensitive data compromised on the cloud, such as clients’ credit card information, as the public cloud may be vulnerable to privileged database administrators and hackers who may gain accesses to disk-resident data, or learn from the query results requested by DO.

*SDB*[1] is a secure query processing system that supports *data interoperability* with secret sharing encryption scheme, where the output of an operator can be taken as input of another operator, which leads to a wide range of SQL queries executable by cloud database, without revealing sensitive information. Besides security, we also demonstrate that SDB is practically efficient.

In this project, we implemented prototype of SDB by query rewriting, on top of Apache Spark. Specifically, this version of SDB supports the following kind of data interoperable query operations:

- Encryption for integer type data
- Secure operators: addition, subtraction, multiplication, comparison
- Secure aggregation function: count
- Web interface for table creation, data uploading and secure querying
2. Project Background and Literature Review

There exist many approaches tackling the challenge of data security. One common way is to encrypt data before uploading to the cloud server. For instance, Oracle database 11g provides Transparent Data Encryption to encrypt disk-resident data, as a service of reliable data storage and administration. However, as the data is not preserved after conventional encryption, query processing on encrypted data is impossible and the computational power of cloud service provider is thus lost. To execute queries, DO has to adopt a *Decrypt-Before-Query (DBQ)* approach, that is, shipping all ciphertext to DO, decrypting them before executing queries.

On the other hand, *fully homomorphic encryption* (FHE) techniques are developed to realize computation on encrypted data. Despite its theoretical significance, such encryption scheme is too computationally expensive for practical use, not to mention data-intensive queries on the cloud. For example, a recent implementation of FHE[2] lets us process 180 AES blocks in around 18 minutes using 3.7GB RAM, whose running time is far from satisfactory for industry use.

Several partially homomorphic encryption schemes are proposed to process particular types of operations on encrypted data, such as OPES[3](which supports homomorphic comparison encryption) and RSA(which supports homomorphic multiplication encryption). However, a simple selection query such as “\texttt{uni-cost * quantity < budget}” cannot be processed on encrypted data by piecing different encryption functions together.

TrustedDB[4] and Cipherbase are hardware-based secure processors, which use secure hardware components to store private keys of sensitive data during processing. However, since security is
dependent on the specific hardware, such secure processors cannot scale up as clusters easily and have to be replaced in case of hardware failure, which makes them not economically scalable.

CryptDB[5] is a well-known system for processing encrypted data, using an onion encryption approach, implemented upon MySQL. CryptDB uses different levels of encryptions to support different security strength and computational support. For example, it uses RSA to allow equality check on encrypted data, and uses OPES to achieve ordering preserving comparison. Under CryptDB, data security is gradually released to a level to achieve appropriate computation requirement. However, some operators of CryptDB are not secure against chosen plaintext attack, and it’s capable of a limited range of secure operations.

SDB is a secure query processing system for relational database, using secret sharing encryption schema. With the data interoperable secure operators supported by secret sharing encryption, SDB simulates fully homomorphic encryption for sensitive data stored in database, which leads to a wide range of queries supported by cloud database server. A typical query “SELECT name, salary + 1000 FROM employee WHERE salary * 12 > 100000 AND age < 25” (with salary being sensitive), which requires additive, multiplicative and order-preserving homomorphism, is supported by SDB. Such query will be executed at cloud server first, afterwards data owner receives the encrypted query result (decrypted values of salary + 1000 and plaintext of name) and decrypts it to get the original result. This way, SecureDB turns out to be both secure and efficient.
3. Project Methodology

3.1 Theoretical background

3.1.1 Encryption Procedure

SDB encrypts sensitive data with a secret sharing method per column. The DO maintains the secret keys of columns while SP maintains the encrypted value of sensitive columns, plaintext of non-sensitive columns together with some crypto helper columns.

To encrypt data with secret sharing, the DO first generate two secret numbers, g and n. n is the product of two big random prime numbers $\rho_1, \rho_2$, while g is a positive number that is co-prime with n. We define

$$n = \rho_1 \rho_2$$

$$\phi(n) = (\rho_1 - 1)(\rho_2 - 1).$$

Consider a sensitive column A to be encrypted. DO generates a pair of column keys $<m, x>$ for column A randomly. Also, for each row of column A, DO generates a distinct row-id randomly. We require that $0 < r, m, x < n$.

We use $[v]$ to denote a sensitive value to be encrypted, $v_e$ to denote the ciphertext after encryption, $v_k$ to denote the item key of $[v]$.

The item key is generated by

$$v_k = gen(r, (m, x)) = mg^{(rx \mod \phi(n))} \mod n.$$  

The encrypted value is computed by

$$v_e = E([v], v_k) = [v]v_k^{-1} \mod n$$

To recover plaintext from ciphertext, we compute

$$[v] = D(v_e, v_k) = v_e v_k \mod n$$

Row-id is encrypted by additive homomorphic encryption E().
The encryption procedure is illustrated in Figure 2. It shows how sensitive data is transformed into encrypted values. To reveal the plaintext, DO only needs to store the column keys, while SP maintains the bulk encrypted data.

![Encryption Procedure Diagram]

**Figure 2: Encryption procedure \((g = 2, \ n = 35)\).**

### 3.1.1 Secure Operators

Data interoperability is one of the most important qualities of SDB. Table 1 shows a list of primitive secure operators implemented in current version of SDB.

<table>
<thead>
<tr>
<th>Operator</th>
<th>expression</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\times)</td>
<td>(A \times B)</td>
<td>vector dot product of two columns of the same table</td>
</tr>
<tr>
<td>(+, -)</td>
<td>(A + B, A - B)</td>
<td>vector addition/subtraction of two columns of the same table</td>
</tr>
<tr>
<td>=</td>
<td>(A = B)</td>
<td>equality comparison on two columns of the same table and output a binary column of ‘0’ and ‘1’</td>
</tr>
<tr>
<td>&gt;</td>
<td>(A &gt; B)</td>
<td>ordering comparison on two columns of the same table and output a binary column of ‘0’ and ‘1’</td>
</tr>
<tr>
<td>(\pi)</td>
<td>(\pi S(R))</td>
<td>project table R on attributes specified in an attribute set S (\otimes)</td>
</tr>
<tr>
<td>Count</td>
<td>Count(R)</td>
<td>count the number of rows in a relation</td>
</tr>
</tbody>
</table>

**Table 1: list of primitive secure operators**
Notice that SDB supports data operability in different modes. For instance, the \( \times \) operator, can be applied in three scenarios: operands are both encrypted\((EE \ Mode)\); one operand is encrypted, the other is a constant\((EC \ Mode)\); one operand is encrypted, the other is plain\((EP \ Mode)\). Besides row-id, column R(with encrypted values of random number) and column S(with encrypted values of 1’s) help achieve secure operation.

![Figure 4: \( C = A \times B \ (g = 2, \ n = 35) \).](image)

The protocol for secure multiplication in EE mode, and an illustrative figure are listed above. To calculate the secure multiplication between two sensitive columns, DO simply generates a new column key while SP performs expensive computation on bulk data. The encrypted result is then sent back to DO and decrypted by the new column key. The proof for correctness and security can be found in [1].

*Key update* is a helper operator which takes an column A and a target column key \(<m_c, x_c>\) and output a column C that shares the same plaintext of A with target column key, without revealing sensitive information. Other protocols of secure operators implemented in SDB are also listed below.
Data: Column $A$ with column key $\langle m_A, x_A \rangle$ and a constant $p$

Result: $C = pA$ with $C$’s column key $\langle m_C, x_C \rangle$

Client-protocol:
\[ x_C = x_A; \]
\[ m_C = pm_A \mod n; \]
Indicate that $C$’s encrypted column is $A$’s encrypted column;

Server-protocol:
Nil

Algorithm 2: EC multiplication

Data: Column $A$, $B$ with column key $\langle m_A, x_A \rangle$ and $\langle m_B, x_B \rangle$

Result: $C = A + B$ with $C$’s column key $\langle m_C, x_C \rangle$

Client-protocol:
Generate random $m_C, x_C$
\[ A' = \kappa(A, c_kC); /\text{ DO executes client-protocol} \]
\[ B' = \kappa(B, c_kC); /\text{ of key update}. \]
Set $C$’s column key as $\langle m_c, x_c \rangle$;

Server-protocol:
\[ A' = \kappa(A, c_kC); /\text{ SP executes server-protocol} \]
\[ B' = \kappa(B, c_kC); /\text{ of key update}. \]
for each row $r$ do
  Let $a_r'$, $b_r'$ be the encrypted values on $A'$, $B'$;
  Set encrypted value of $C$ $c_r = a_r' + b_r' \mod n$;
end

Algorithm 4: EE addition/subtraction

Data: (i) Column $A$ with column key $\langle m_A, x_A \rangle$; (ii) target column key $\langle m_C, x_C \rangle$

Result: $C = A$ with $C$’s column key $\langle m_C, x_C \rangle$

Client-protocol:
Let $\langle m_B, x_B \rangle$ be the column key of $S$;
\[ p = x_B^{-1}(x_C - x_A) \mod \phi(n); \]
\[ q = m_A m_B^{-1} \mod n; \]
Send $p, q$ to SP;
Set $C$’s column key as $\langle m_C, x_C \rangle$;

Server-protocol:
Obtain $p, q$ from DO;
for each row $r$ do
  Let $a_r, s_r$ be the encrypted values on $A, S$;
  Set encrypted value of $C$ $c_r = qa_r s_r \mod n$;
end

Algorithm 3: Key update

Data: Column $A$, $B$ with column key $\langle m_A, x_A \rangle$ and $\langle m_B, x_B \rangle$

Result: A column of comparison results $C$: $0$ if $A = B$; $1$ if $A > B$; return $-1$ if $A < B$

Client-protocol:
\[ Z = R(A - B); /\text{ EE addition, EE multiplication} \]
\[ Z' = \kappa(Z, (1,0)); /\text{ DO executes client-protocol} \]

Server-protocol:
\[ Z = R(A - B); /\text{ Corresponding server protocol} \]
\[ Z' = \kappa(Z, (1,0)); /\text{ Corresponding server protocol} \]
for each row $r$ do
  Let $z_r'$ be the values on $Z'$;
  switch $z_r'$ do
    case $0$
      \[ c_r = 0; /\text{ is the result of this row} \]
    case $> 0$
      \[ c_r = 1; \]
    case $< 0$
      \[ c_r = -1; \]
  end
end

Algorithm 5: Comparison
3.2 System Architecture

3.2.1 Overall Architecture

SDB is implemented as a software-layer on top of *Apache Spark*. Apache Spark is a fast and general engine for large-scale data processing, with in-memory primitives that enhance fast cluster computing. *Apache Hadoop* serves as the cluster manager and distributed storage system for very large data sets to support Spark’s cluster computation. *Apache Hive* is the distributed data warehouse to store data.

*Spark SQL* is a component on top of Spark Core that provides support for structured data and SQL support. Spark SQL supports *User-Defined Functions*, which provides a mechanism for extending the functionality of the cluster database server by adding cryptographic function that can be evaluated in Spark SQL. This way, SDB’s server protocol is pushed to Spark execution engine and executed as UDFs.

The SDB Proxy resides on client side and stores the secret keys in the *key store*. A typical life cycle of a query consists of 5 stages:

1. Application submits a query to SDB proxy.
2. SDB proxy parses, analyses and rewrites the query with SDB UDFs.
3. SDB proxy sends the rewritten query to Spark SQL.
4. Spark SQL evaluates the query and sends the encrypted result to SDB Proxy.
5. SDB Proxy decrypts and returns the result to the application.

![Figure 5: Architecture of SDB](image)
3. SDB proxy submits the rewritten query to Spark SQL for server protocol execution.
4. Spark SQL sends back the encrypted query result to SDB proxy.
5. SDB proxy decrypts the query result and sends it back to application.

The advantages of SDB’s architecture include

1. Performance wise
   • Secure operators are implemented as UDF which are executed in the same address space of the
     SparkSQL, which leads to less memory copy, less network transfer and no IPC.

2. Engineering wise
   • SDB leverages the SparkSQL’s optimizer to optimize the server side queries.
   • SDB leverages all the normal operators provided by SparkSQL.
   • Machine failures of the working nodes, disk-based processing and parallelism are well taken
     care of by Spark.

3.2.2 System Components

SDB proxy is composed of 4 major components:

1. SDB Query Parser
2. SDB Semantic Analyser
3. SDB Query Rewriter
4. SDB Query Executor

Figure 7: Components of SDB Proxy
**SDB Query Parser** parses a query string and transforms it into an abstract syntax tree (AST). It is built based on HiveQL Parser, with additional tokens for column sensitivity declaration. For example, to declare a table T with sensitive column A and non-sensitive column B, the user has to submit query “CREATE TABLE T (A int ENC, B int)”, which is transformed into the AST in figure 8.

![Figure 8: AST of “CREATE TABLE T (A int ENC, B int)”](image)

**SDB Semantic Analyzer** transforms an AST into a Logical Plan Tree (LPT) as the preparation for query rewrite. The semantic analyser traverses the AST to construct a LPT, which is easier to manipulate with, and access the key store to

1. verify whether the reference of queried tables and columns are valid
2. annotate a column with its sensitivity and its column key, if any

![Figure 9: Transformation from AST to LPT](image)
**SDB Query Rewriter** rewrites the LPT with secure operator, and constructs a *Physical Plan Tree (PPT)*. While traversing LPT, every time the rewriter identifies a normal arithmetic operator involving sensitive columns, it replaces the normal operator with the corresponding secure operator, based on SDB rewrite rule (discussed in detail in next section). Finally, the rewrite constructs a physical plan tree for SDB Executor to execute.

![Figure 9: Transformation from LPT to PPT](image)

**SDB Executor** executes the physical query plan and returns the result set. Finally, the executor executes the physical plan tree which normally splits the execution into two phases. In the first phase, it sends server-side query and waits for query result. In the second phase, it receives server-side query and decrypts query result. The decrypted result is then sent back to client application via connector.
3.3 Query Rewriting

We denote $ck_A = <m_A, x_A>$ as the column key for column A, denote $\kappa_{AC} = <p, q>$ as the client-side result of key update column A with target column key $<m_C, x_C>$. Besides, we use $ck_Z$ to denote the special column key $<1, 0>$.

### 3.3.1 Query Rewriting for Multiplication

According to the secret sharing scheme, query rewriting for secure multiplication is straightforward.

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand($\rho_1, \rho_2$)</td>
<td>Client-side function which generates a random number co-prime with $n, \Phi(n)$</td>
</tr>
<tr>
<td>key_update_client($ck_A, ck_C, kks, \rho_1, \rho_2$)</td>
<td>Client-side key update which computes $\kappa_{AC}$</td>
</tr>
<tr>
<td>decrypt($Ce, ck_C, n, g, row_id$)</td>
<td>Client-side decryption which computes $[c]$</td>
</tr>
<tr>
<td>sdb_mul($Ae, Be, n$)</td>
<td>Server-side multiplication $(A \times B)$ UDF</td>
</tr>
<tr>
<td>sdb_add($Ae, Be, Se, \kappa_{AC}, \kappa_{BC}, n$)</td>
<td>Server-side addition $(A + B$ as C) UDF</td>
</tr>
<tr>
<td>sdb_compare($Ae, n$)</td>
<td>Server-side comparison $(A &gt; 0)$ UDF</td>
</tr>
<tr>
<td>sdb_key_update($Ae, Se, \kappa_{AC}, n$)</td>
<td>Server-side key update UDF</td>
</tr>
</tbody>
</table>

Table 2: List of functions for query rewriting

#### 3.3.2 Query Rewriting for Addition

For secure addition in EE Mode, addition is achieved by applying key update to both A and B so that they share the same column key.

<table>
<thead>
<tr>
<th>Input: SELECT $A \times B$ as C FROM T</th>
<th>Input: SELECT $A \times u$ as C FROM T</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ck_C = &lt;m_A \times m_B, x_A + x_B&gt;$</td>
<td>$ck_C = &lt;m_A \times u, x_A&gt;$</td>
</tr>
<tr>
<td>Output: SELECT row_id, sdb_mul($Ae, Be, n$) AS $Ce$ FROM T</td>
<td>Output: SELECT row_id, $Ae$ AS $Ce$ FROM T</td>
</tr>
</tbody>
</table>

Rule 1: Rewrite Rule for Multiplication (EE & EC mode)
For secure addition in EC mode, the expression $A+u$ can be considered as $A+(S \times u)$, as the plaintext of $S$ is 1. Thus an EC addition operation can be done as an EE addition after EC Multiplication.

**Rule 2: Rewrite Rule for Addition (EE mode)**

```sql
Input: SELECT A + u as C FROM T

ck_u = <u \times mS, xS>
ck_c = <rand(p1, p2), rand(p1, p2)>
\kappa_{ac} = key\_update\_client(ckA, ck_c, cks, p1, p2)
\kappa_{ac} = key\_update\_client(ckAs, ckC, cks, p1, p2)

Output: SELECT row_id, sdb_add(A_s, B_s, S_s, \kappa_{ac}, \kappa_{ac}, n) AS C_e FROM T
```

3.3.3 Query Rewriting for Subtraction

For secure subtraction in EE Mode, the expression $A-B$ can be considered as $A+ (B \times (-1))$, similar to addition in EC mode. Based on the modular equation $-1 \mod n = (n - 1) \mod n$, we compute the column key of $-B$ as follows.

**Rule 3: Rewrite Rule for Addition (EC mode)**

```sql
Input: SELECT A + u as C FROM T

ck_{Bu} = <u \times mS, xS>
ck_c = <rand(p1, p2), rand(p1, p2)>
\kappa_{ac} = key\_update\_client(ckA, ck_c, cks, p1, p2)
\kappa_{ac} = key\_update\_client(ck_{Bu}, ckC, cks, p1, p2)

Output: SELECT row_id, sdb_add(A_s, B_s, S_s, \kappa_{ac}, \kappa_{ac}, n) AS C_e FROM T
```

For secure subtraction in EC mode, the expression $A-u$ can be considered as $A-S \times u$, thus we simply compute based on the rule of EE subtraction.

**Rule 4: Rewrite Rule for Subtraction (EE mode)**

```sql
Input: SELECT A - B as C FROM T

ck_{Bi} = <(n-1) \times mS, xB>  // inverse the value of B
ck_c = <rand(p1, p2), rand(p1, p2)>
\kappa_{ac} = key\_update\_client(ckA, ck_c, cks, p1, p2)
\kappa_{ac} = key\_update\_client(ck_{Bi}, ckC, cks, p1, p2)

Output: SELECT row_id, sdb_add(A_s, B_s, S_s, \kappa_{ac}, \kappa_{ac}, n) AS C_e FROM T
```
3.3.4 Query Rewriting for Comparison

For secure comparison in EC mode, the expression \( A > B \) can be considered as \( (A - B) > 0 \), to ensure no sensitive information is leaked, we randomize the result by \( R \times (A - B) > 0 \) at server side so that the result of \( A - B \) is not revealed.

**Input:** SELECT A FROM T WHERE A > B

\[
\begin{align*}
ck_B &= <(n-1) \times ms, xB> \quad \text{// inverse the value of } B \\
ck_c &= <\text{rand}(p1, p2), \text{rand}(p1, p2)>
\end{align*}
\]

\( \kappa_{AC} = \text{key_update_client}(ck_A, ck_c, cks, p1, p2) \)

\( \kappa_{BC} = \text{key_update_client}(ck_{Bi}, ck_c, cks, p1, p2) \quad \text{// } A - B \\
ck_R &= <mR \times mC, xB + xC> \quad \text{// } R \times (A - B) \\
\kappa_{BCZ} &= \text{key_update_client}(ck_{RC}, ck_z, cks, p1, p2)
\]

**Output:** SELECT A, row_id FROM T

WHERE \( \text{sbdb_compare}(	ext{sbdb_key_update}(	ext{sbdb_mul}(Rs, \text{sbdb_add}(Ae, Be, S_e, \kappa_{AC}, \kappa_{BC}, n)), S_e, \kappa_{BCZ}, n), n) > 0 \)

**Rule 6: Rewrite Rule for Comparison (EE mode)**

For secure comparison in EC mode, the expression \( A > U \) can be considered as \( (A - S \times U) > 0 \). Notice that we give the rewriting rule for >, to apply it to = or <, simply swap > with the target equality operator.

**Input:** SELECT A FROM T WHERE A > u

\[
\begin{align*}
ck_{Su} &= <u \times ms, xs> \quad \text{// } S \times u \\
ck_{Sui} &= <(n-1) \times msu, xsu> \quad \text{// } -(S \times u) \\
ck_c &= <\text{rand}(p1, p2), \text{rand}(p1, p2)>
\end{align*}
\]

\( \kappa_{AC} = \text{key_update_client}(ck_A, ck_c, cks, p1, p2) \)

\( \kappa_{Sui} = \text{key_update_client}(ck_{Sui}, ck_c, cks, p1, p2) \quad \text{// } A - S \times u \\
ck_{RC} &= <mR \times mC, xB + xC> \quad \text{// } R \times (A - B) \\
\kappa_{BCZ} &= \text{key_update_client}(ck_{RC}, ck_z, cks, p1, p2)
\]

**Output:** SELECT A, row_id FROM T

WHERE \( \text{sbdb_compare}(	ext{sbdb_key_update}(	ext{sbdb_mul}(Rs, \text{sbdb_add}(Ae, S_e, S_e, \kappa_{AC}, \kappa_{Sui}), S_e, \kappa_{BCZ}, n)), S_e, \kappa_{BCZ}, n), n) > 0 \)

**Rule 7: Rewrite Rule for Comparison (EC mode)**
Figure 10 shows the rewritten physical plan tree of EE Comparison query.

Figure 10: Rewritten LPT of SELECT A FROM T WHERE A > B
3.4 Web Interface

3.4.1 Web Interface for Uploading Data

We introduce the steps to upload a table of sensitive data as follows.

1. In the *SQL Editor* page, type in the CREATE TABLE query

2. Append keyword “ENC” to every sensitive field

3. Click “Execute SQL Query” button

![Figure 11: Web interface to create a table with two sensitive fields (A, B)](image)

4. Receive confirmation message “Create Table Successful” after success

![Figure 12: Confirmation message of table creation success](image)

5. In the *Upload Data* page, select table name and data source file to upload.

6. Click “Upload Data” button to start the upload process.
3.4.2 Web Interface for Query

We introduce the interface for query as follows.

1. Type in the query in the SQL Editor page
2. Click “Execute SQL Query” button
3. Query result will be displayed as in Figure 14

Figure 13: Web interface to upload data to table

Figure 14: Web interface to execute query
4. At “Execution Analysis” section, the query executed at cluster server is displayed.

5. Client and server side query execution cost are visualized in charts.

Cloud service provider’s view of encrypted sensitive data is illustrated in Figure 16.
4. Experiments and Results

4.1 Experiment Environment

The prototype of SDB is deployed on a cluster of 10 machines. Every server machine is equipped with eight Intel(R) Core(TM) i7-3770 CPUs, four 4GB Kingston DDR3 1333MHz RAM, running on Ubuntu Release 12.04. We use Apache Hadoop 2.4.1, Spark 1.1.0 and Hive 0.12.0, running on Java 1.6.0. The Hadoop File System is configured to have 3 replication of files and 64 MB data blocks. Spark executors are configured to have memory of 256 MB. The SDB proxy resides on the client machine with a single Intel 2.40 GHz Core i5 CPU.

In our experiment, we answer the question of Q1: “How important it is for the secure database operators to be data interoperable in a cloud database environment?” Second, we address the questions of Q2: “How much performance overhead is introduced by SDB to protect sensitivity? Is it practical in query processing?”

We prepare a synthetic table T with two sensitive columns A and B. The values of each column are generated with a uniform distribution. We execute 3 kinds of queries:

- **[Range Query EE Mode]** SELECT A, B from T WHERE A < q
- **[Range Query EC Mode]** SELECT A, B from T WHERE A < B
- **[Count Query]** SELECT count(A) from T WHERE A < q

The parameter q controls the selectivity of a range query. A smaller q leads to a smaller result size.

We split the total query time into two components. (1) *Server Cost*: the time taken for the cluster server to execute rewritten queries. (2) *Client Cost*: the time taken for SDB proxy to parse, analyse, rewrite queries and decrypt query results.
4.2 Experiment Result

4.2.1 SDB vs. DBQ

To answer Q1, we compare the performance of SDB with Decrypt-Before-Query Model (DBQ). DBQ is configured with the same cluster machines at server-side and runs Spark standalone mode at client side. Since no computation is done on sensitive data at server side, sensitive columns are encrypted using RSA encryption. For a sample query “SELECT A, B from T WHERE A < q”, DBQ’s server cost is simply the time taken to ship all the data back to client. DBQ client then decrypts all sensitive data to execute the range query. As shown in Figure 17, the query processing time of DBQ is dominated by client cost, while SDB is dominated by server cost. We conclude that the client cost of SDB is significantly less than that of DBQ.

4.2.2 SDB Cost Breakdown

We further break down the components of SDB cost as follows. Figure 18 shows the client/server cost of a range EC query (on 200K dataset). With increasing selectivity, server cost stays stable, while client cost increases linearly. Figure 19 shows the further detail of client cost of a range EC query (on 5K dataset). This shows that decryption cost increases as the number of results to decrypt increases. Decryption cost is the major contributor to client cost. We observe that the decryption cost is significant, which may become the bottleneck of query execution when result size is large. Therefore, improving the performance of decryption is one of the key optimizations to be done in the future.
4.2.3 SDB vs. SparkSQL

To answer Q2, we evaluate the performance of SDB as follows. We first execute the three queries on SparkSQL directly with all columns stored as plaintext values, bypassing all secure operators. The time to execute under this scenario is denoted as $T_{\text{SparkSQL}}$. Then we execute the same queries on encrypted data on SDB, and use $T_{\text{SDB}}$ to denote the execution time. The ratio $T_{\text{SDB}} / T_{\text{SparkSQL}}$ captures the degree to which SDB’s encryption and secure protocol slows down the entire query processing time.

We observe that SDB introduces a 170 times slowdown of the query processing time. This is mainly the result of expensive modular exponent computation of big integers. In particular, the secure comparison involves exponent computation at least three times. This forms another aspect of future optimization we will focus on.

![Figure 18: Client/Server Cost vs. Selectivity](image1)
![Figure 19: Client Costs vs. Selectivity](image2)
![Figure 20: T_{\text{SDB}} / T_{\text{Spark}}](image3)
Finally, Figure 18 shows the client/server cost for the three queries with different data size.

Figure 21 Client/Server Cost vs. Data Size
5. Conclusion and Future Works

The future work of SDB is mainly divided into two parts: secure operator extension and cryptographic optimization.

5.1 Secure Operator Extension

SDB is designed to support a wide range of query with several secure operators. In Appendix 7.1, we included the query rewriting rules for the following secure operators which are not fully implemented yet:

- Secure multiplication, addition, subtraction, comparison in EP mode
- Secure cartesian product
- Secure aggregation function: sum(), avg()

Since the query executor of SDB proxy is implemented with the iterator pattern, SDB proxy can also be extended to adopt the split client/server execution approach proposed by MONOMI[6]. This way, in the case of un-supported queries, SDB proxy shall identify the parts of query supported at cluster server and push them to server as much as possible. Then SDB proxy performs the rest of the queries.

5.2 Cryptographic Optimization

As we mentioned in the experiment, the decryption cost dominates other costs at client side. Therefore, it’s crucial to improve the performance of decryption. During the computation of the item key of the same column but different rows, it can be computed as \( mb^r \mod n \) with \( b = g^x \). Thus it can be viewed as exponential computation with the same base, which can be optimized by Exponentiation by squaring[7]. To optimize such computation, we first pre-compute a
table of \( b^1, b^2, b^4, b^8, b^{16}, \ldots \) To compute \( b^{13} \), we only need to retrieve \( b^1, b^2, b^8 \) from table and multiply them together.

At server side, currently User Defined Functions at server side are all written in Java, whose modular exponential computation is slow. Instead, we can achieve 4x better performance[8] by calling native math methods in GMP library, which is written in C++.

Besides, there are studies on secure indexing techniques[9], which groups the value of sensitive columns into coarse ranges. Such index filtering can be applied before we process a query using our secure operators.

5.3 Conclusion

In conclusion, we gave a comprehensive description of SDB’s query rewriting and analysis of performance. By deploying such secret sharing scheme that supports multiple secure operators with data interoperability, SDB is a query processing system that is both secure and practically efficient.
6. References


7. Appendix

7.1 Rewrite Rules for Other Secure Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIES(row_id, Ck_ROW_ID, (\rho_1, \rho_2))</td>
<td>Client-side function which decrypts row_id with SIES</td>
</tr>
<tr>
<td>sdb_int_add(row_id1, row_id2, n)</td>
<td>Server-side row-id addition UDF</td>
</tr>
<tr>
<td>sdb_cartprod(Ae, Sc, pA, n)</td>
<td>Server-side cartesian product ((A \otimes B) as C) UDF</td>
</tr>
</tbody>
</table>

Secure operation in EP mode is similar to EC mode that plaintext is treated as a column with column key \(<1, 0>\).

\[
\begin{align*}
\text{SELECT A \times P as C FROM T} \\
\quad c_{k_{PN}} &= <\text{rand}(\rho_1, \rho_2), \text{rand}(\rho_1, \rho_2)> \\
\quad k_{ZPN} &= \text{key}._{\text{update}}._{\text{client}}(ck_z, c_{k_{PN}}, c_{k_S}, \rho_1, \rho_2) \\
\quad c_{k_C} &= <m_A \times m_{PN}, x_A + x_{PN}>
\end{align*}
\]

\[
\text{SELECT row_id, sdb\_mul(Ae, sdb\_key\_update}(P, S_e, k_{ZPN}, n), n)\text{ AS C}_e\text{ FROM T}
\]

**Rule 8: Rewrite Rule for Multiplication (EP Mode)**

\[
\begin{align*}
\text{SELECT } (A+P) \text{ as C FROM T} \\
\quad c_{k_{PN}} &= <\text{rand}(\rho_1, \rho_2), \text{rand}(\rho_1, \rho_2)> \\
\quad k_{ZPN} &= \text{key\_update\_client}(ck_z, c_{k_{PN}}, c_{k_S}, \rho_1, \rho_2) \\
\quad c_{k_{PC}} &= <\text{rand}(\rho_1, \rho_2), \text{rand}(\rho_1, \rho_2)> \\
\quad k_{AC} &= \text{key\_update\_client}(ck_a, c_{k_C}, c_{k_S}, \rho_1, \rho_2) \\
\quad k_{PCN} &= \text{key\_update\_client}(ck_{PC}, c_{k_C}, c_{k_S}, \rho_1, \rho_2)
\end{align*}
\]

\[
\text{SELECT row_id, sdb\_add(Ae, sdb\_key\_update}(P, S_e, k_{ZPN}, n), S_e, k_{AC}, k_{PCN}, n)\text{ AS C}_e\text{ FROM T}
\]

**Rule 9: Rewrite Rule for Addition (EP Mode)**

\[
\begin{align*}
\text{SELECT } (A-P) \text{ as C FROM T} \\
\quad c_{k_{PN}} &= <\text{rand}(\rho_1, \rho_2), \text{rand}(\rho_1, \rho_2)> \\
\quad k_{ZPN} &= \text{key\_update\_client}(ck_z, c_{k_{PN}}, c_{k_S}, \rho_1, \rho_2) \\
\quad c_{k_{PN}} &= <(n-1) \times m_{PN}, x_{PN}> \quad // \text{inverse the value of } P \\
\quad c_{k_{C}} &= <\text{rand}(\rho_1, \rho_2), \text{rand}(\rho_1, \rho_2)> \\
\quad k_{AC} &= \text{key\_update\_client}(ck_a, c_{k_C}, c_{k_S}, \rho_1, \rho_2) \\
\quad k_{PCN} &= \text{key\_update\_client}(ck_{PC}, c_{k_C}, c_{k_S}, \rho_1, \rho_2)
\end{align*}
\]

\[
\text{SELECT row_id, sdb\_add(Ae, sdb\_key\_update}(P, S_e, k_{ZPN}, n), S_e, k_{AC}, k_{PCN}, n)\text{ AS C}_e\text{ FROM T}
\]

**Rule 10: Rewrite Rule for Subtraction (EP Mode)**
For secure Cartesian product between two tables, we make use of the additive homomorphic property of row_id's encryption (SIES) so that T1 JOIN T2 can be decrypted with the new row_id.

\[
C = mz \times Ce
\]

Rule 11: Rewrite Rule for Comparison (EP Mode)

For secure Cartesian product between two tables, we make use of the additive homomorphic property of row_id's encryption (SIES) so that T1 JOIN T2 can be decrypted with the new row_id.

Rule 12: Rewrite Rule for Cartesian Product

To perform secure aggregation function SUM(), we perform a key update operation on target column A with a special column key < rand, 0 >. Particularly, we insert with a different row_id in case the result of SUM() is referred in other nested queries. Finally, The decryption of SUM ciphertext is handled differently by equation \[ C = mz \times Ce \]

Rule 13: Rewrite Rule for SUM()
SELECT avg(A) as C FROM T

CkSUM = < rand(p1, p2), 0 >
\kappa_{\text{SUM}} = \text{key_update_client}(ck_A, ck_{\text{SUM}}, \text{ck}, p1, p2)
ck_{\text{PRI}} = (n-1) \times m_{\text{PN}}, x_{\text{PN}} >
\text{row}_\text{id} = \text{rand}(p1, p2)
\text{enc_row}_\text{id} = \text{SIES(row}_\text{id}, Ck_{\text{ROW}_\text{id}}, p1, p2)

SELECT \text{enc_row}_\text{id}, \text{sdb_sum}(\text{sdb_key_update}(Ae, Se, \kappa_{\text{SUM}}, n)) \text{ as sum, count(Ae) as count FROM T}

Rule 14: Rewrite Rule for AVG()

7.2 Experiment Data

<table>
<thead>
<tr>
<th></th>
<th>5K</th>
<th>200K</th>
<th>400K</th>
<th>600K</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDB Server</td>
<td>4.246</td>
<td>46.934</td>
<td>89.619</td>
<td>132.614</td>
</tr>
<tr>
<td>SDB Client</td>
<td>0.082</td>
<td>2.107</td>
<td>4.372</td>
<td>6.366</td>
</tr>
<tr>
<td>DBQ Server</td>
<td>0.230</td>
<td>8.93</td>
<td>17.860</td>
<td>26.8</td>
</tr>
<tr>
<td>DBQ Client</td>
<td>14.33</td>
<td>510.92</td>
<td>1003.931</td>
<td>1483.658</td>
</tr>
</tbody>
</table>

Figure 17: SELECT A, B from T WHERE A < p, 1% selectivity (seconds)

<table>
<thead>
<tr>
<th></th>
<th>0.01%</th>
<th>1%</th>
<th>2%</th>
<th>3%</th>
<th>4%</th>
<th>5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Cost</td>
<td>44.239</td>
<td>44.327</td>
<td>45.035</td>
<td>45.502</td>
<td>45.627</td>
<td>45.714</td>
</tr>
<tr>
<td>Client Cost</td>
<td>0.071</td>
<td>2.107</td>
<td>4.048</td>
<td>5.791</td>
<td>7.789</td>
<td>9.835</td>
</tr>
</tbody>
</table>

Figure 18: SELECT A, B from T WHERE A < p, on 200K records (seconds)

<table>
<thead>
<tr>
<th></th>
<th>0.01%</th>
<th>1%</th>
<th>2%</th>
<th>3%</th>
<th>4%</th>
<th>5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse Cost</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Analyse Cost</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>Rewrite Cost</td>
<td>15</td>
<td>14</td>
<td>18</td>
<td>14</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>Decrypt Cost</td>
<td>2</td>
<td>56</td>
<td>104</td>
<td>159</td>
<td>215</td>
<td>253</td>
</tr>
</tbody>
</table>

Figure 19: SELECT A, B from T WHERE A < p, on 5K records (milliseconds)
<table>
<thead>
<tr>
<th></th>
<th>EC Range</th>
<th>EE Range</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Cost</td>
<td>44827</td>
<td>44867</td>
<td>43728</td>
</tr>
<tr>
<td>Client Cost</td>
<td>2107</td>
<td>1786</td>
<td>35</td>
</tr>
<tr>
<td>SparkSQL Cost</td>
<td>250</td>
<td>253</td>
<td>254</td>
</tr>
</tbody>
</table>

Figure 20: TSDB vs TSparkSQL on 200K records (milliseconds)

<table>
<thead>
<tr>
<th></th>
<th>5K</th>
<th>200K</th>
<th>400K</th>
<th>600K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Cost</td>
<td>4.633</td>
<td>44.867</td>
<td>84.053</td>
<td>134.601</td>
</tr>
<tr>
<td>Client Cost</td>
<td>0.14</td>
<td>1.786</td>
<td>3.414</td>
<td>5.358</td>
</tr>
</tbody>
</table>

Figure 20A: SELECT A, B from T WHERE A < B, 1% selectivity (seconds)

<table>
<thead>
<tr>
<th></th>
<th>5K</th>
<th>200K</th>
<th>400K</th>
<th>600K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Cost</td>
<td>4.164</td>
<td>44.827</td>
<td>85.247</td>
<td>126.248</td>
</tr>
<tr>
<td>Client Cost</td>
<td>0.082</td>
<td>2.107</td>
<td>4.372</td>
<td>6.366</td>
</tr>
</tbody>
</table>

Figure 20B: SELECT A, B from T WHERE A < p, 1% selectivity (seconds)

<table>
<thead>
<tr>
<th></th>
<th>5K</th>
<th>200K</th>
<th>400K</th>
<th>600K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Cost</td>
<td>4.45</td>
<td>43.728</td>
<td>82.991</td>
<td>122.435</td>
</tr>
<tr>
<td>Client Cost</td>
<td>0.017</td>
<td>0.035</td>
<td>0.021</td>
<td>0.021</td>
</tr>
</tbody>
</table>

Figure 20B: SELECT COUNT(A) from T WHERE A < p, 1% selectivity (seconds)